08:30:40 OCA PAD AMENDMENT - PROJECT HEADER INFORMATION 12/05/91

Active

Project #: E-21-558 Cost share #: Rev #: 2
Center # : 10/11-6-06296-0A0 Center shr #: OCA file #:
Work type : INST
Contract#: I2571J Mod #: MEMO OF 911125 Document : OTH
Prime #: Contract entity: GIT

Subprojects ? : N CFDA: N/A
Main project #: PE #: N/A

Project unit: ELEC ENGR Unit code: 02.010.118
Project director(s):
SCHIMMEL D E ELEC ENGR (404)853-0077
YALAMANCHILI S ELEC ENGR (404)-

Sponsor/division names: GEORGIA TECH FOUNDATION / CAMPUS
Sponsor/division codes: 400 / 001

Award period: 910901 to 920630 (performance) 920630 (reports)

Sponsor amount	New this change	Total to date
Contract value	0.00	25,000.00
Funded	0.00	25,000.00
Cost sharing amount		0.00

Does subcontracting plan apply ?: N

Title: IBM DEPARTMENTAL GRANT

PROJECT ADMINISTRATION DATA

OCA contact: Kathleen R. Ehlinger 894-4820

Sponsor technical contact

(000)000-0000

Sponsor issuing office

MARK LONG
(404)894-8345

GEORGIA TECH FOUNDATION
CAMPUS

Security class (U,C,S,TS) : U ONR resident rep. is ACO (Y/N): N
Defense priority rating : N/A N/A supplemental sheet
Equipment title vests with: Sponsor GIT X
NONE PROPOSED.
Administrative comments -
ISSUED TO CHANGE PROJECT DIRECTOR FROM S. YALAMANCHILI TO D. E. SCHIMMEL

GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 08/26/92

Project No. E-21-558_____ Center No. 10/11-6-06296-0A0_

Project Director SCHIMMEL D E_____ School/Lab ELEC ENGR_____

Sponsor GEORGIA TECH FOUNDATION/CAMPUS_____

Contract/Grant No. I2571J_____ Contract Entity GIT_

Prime Contract No. _____

Title IBM DEPARTMENTAL GRANT_____

Effective Completion Date 920630 (Performance) 920630 (Reports)

Closeout Actions Required:	Y/N	Date Submitted
Final Invoice or Copy of Final Invoice	N	_____
Final Report of Inventions and/or Subcontracts	N	_____
Government Property Inventory & Related Certificate	N	_____
Classified Material Certificate	N	_____
Release and Assignment	N	_____
Other _____	N	_____

Comments_____

_____

Subproject Under Main Project No. _____

Continues Project No. _____

Distribution Required:

Project Director	Y
Administrative Network Representative	Y
GTRI Accounting/Grants and Contracts	Y
Procurement/Supply Services	Y
Research Property Managment	Y
Research Security Services	N
Reports Coordinator (OCA)	Y
GTRC	N
Project File	Y
Other _____	N
_____	N

Georgia Tech

School of Electrical Engineering

Georgia Institute of Technology
Atlanta, Georgia 30332-0250
Fax 404•853•9171

July 15, 1992

Mr. Paul C. Mugge
ESD Vice President, Systems
IBM Corporation
P.O. Box 1328
Boca Raton, Florida 33429-1328

Dear Mr. Mugge,

Let me take this opportunity to express my appreciation to IBM Corporation for it's continued support of our *Multi Processor PS/2* project. This grant has been an invaluable source of graduate student support in a challenging technical area of mutual interest. We are currently interacting with one systems group within IBM Boca, that is interested in being an active participant in the research supported under this grant.

I would also like to provide you with a report of progress to date. Primarily, the grant has supported the work of Mr. Paul Bond, who has been developing the Microchannel simulator, and Mr. Bruce Kim, who has been working on multiprocessor trace aquisition hardware. These funds have also been used to support several graduate students who are working on Grelated projects during the summer quarter.

I am enclosing a technical report detailing preliminary results of our study which may be of interest to you. Again, a sincere thank you for your generous support, and I look forward to seeing you in the near future. We look forward to continued and mutually benefical interaction with IBM Boca.

Sincerely,

David E. Schimmel
Assistant Professor

cc: Dr. Stanely M. Belyeu
Professor Roger Webb

encl: technical report

Architectural Techniques for the Design of a Multiprocessor PS/2

Paul J. Bond David E. Schimmel
Sudhakar Yalamanchili

Computer Systems Research Laboratory
School of Electrical Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0250

Abstract

While the performance of uniprocessor PC and workstation technology has continued to improve, it is appropriate to consider some degree of parallelism to effect further improvements. We consider a multiprocessor model utilizing shared memory and a single bus. Towards this goal, we must determine the load that an individual processor places on the bus. This load is dependent on both the processor and the application program. These affect the frequency of bus use, including memory reads and updating memory when data is altered. We consider techniques to reduce a processor's dependence on the bus, thus allowing time on the bus for more processors. The presence of shared data in a multiprocessor system necessitates the need to allow all processors access to the most recent value of a shared datum. We simulate multiple processors to evaluate the system's performance. With the measurements given by the simulator, we graph the data and analyze the impact of various parameters. We explore areas for further study in the development of a multiprocessor PS/2 with the IBM Micro Channel architecture.

Contents

1 Introduction

As electrical device characteristics approach physical limits, parallelism is an important alternative for improving computer performance. While most parallel machines have been targeted at relatively high-end markets, the low-end market can also benefit from parallel computing techniques such as multiprocessing. To help provide the user with a good cost/performance ratio, we consider the shared bus as the least complex(therefore, lowest cost) interconnection network for multiple processors. Considering bus limitations, the multiprocessor system must be designed to reduce substantially the communication needs between high-performance microprocessors.

We are interested in studying the ability of generic bus architectures to support a shared-memory model of parallel programming. One technique allowing a bus to accommodate additional processors is the use of coherent private caches. While cache coherence, and related protocols, have been extensively studied, there is usually a reliance on the use of additional hardware mechanisms and signal definitions to implement these policies[3]. Conversely, multiprocessor workstations are often developed using existing bus structures without redefining the bus. We pose a constrained optimization problem to replace an unconstrained: what models of parallel programming, cache consistency, and synchronization are efficiently supported by a specific bus architecture. Clearly, the best solutions to this problem are not necessarily the same as those obtained when we are free to specify extra signal lines and protocols to support our mechanisms. We anticipate that this investigation will provide a useful guide for those attempting to implement such systems in general.

We now enumerate several paradigms which support the concept of multiprocessing using the Micro Channel architecture in particular. First, a single processor can delegate tasks to several highly specialized coprocessors[13]. A second option is for multiple processors to each execute separate applications for individual users. Finally, each processor may contribute work in parallel toward the completion of a single task.

One goal of this research is to determine a limit on the number of useful processors that the micro channel can support. This limit is also a function of cache size and other parameters that aid in reducing overall bus traffic for various applications. Even in the presence of extremely large caches, the number of processors connected to a bus is still limited by compulsory cache misses and actively shared data[7]. Additionally, we are evaluating cache coherence protocols which possess efficient implementations using the micro channel. This evaluation requires assumptions about the granularity of the parallel program. If we let all processors run independent instruction streams, we obtain an upper bound on interprocess performance. The next step is to determine the amount or granularity of parallelism we can effectively use on the micro channel and how that affects the number of useful processors. IBM is also developing designs for processor cards supporting two or more tightly coupled CPUs. One design uses an additional CPU mounted on a daughter-board, while a second design uses two processor cards linked by a cable[5].

The remainder of this paper is organized as follows. In section 2 we describe properties of the Micro Channel architecture, along with our simulated implementation of some features. In section 3 this paper then further discusses the simulator. We present graphs of the simulator output and examine the information they provide in section 4. There are several

features we are considering for future versions of the simulator. We discuss the relevance of these extensions.

2 Background

2.1 The Micro Channel Architecture

The Micro Channel architecture(MCA) consists of signal specifications, timings, physical requirements, and procedural definitions needed for product compatibility[6]. The MCA is used on a wide range of commercial machines and expansion cards[4, 12]. These machines use various combinations of optional and required bus signals according to the architectural specification. The MCA defines several reserved signals for future expansion, along with the option to widen the address and data bus. A stated goal of MCA is card compatibility. Any MCA card which physically fits into a machine should operate correctly, without consideration of the data bus width of the other cards in the machine. A card may implement an optional feature which the host machine is incapable of using. However, the card must not be dependent on that feature for reliable operation[13]. Thus, it is desirable to design a multiprocessor system, implementing a minimum version of MCA, adhering to this philosophy without using signals marked as reserved or otherwise changing the backplane or bus protocols.

The MCA accommodates up to sixteen bus masters in a system, each with a distinct priority level. When a bus master requires access to the bus, it will assert its preempt bus signal. It then participates in arbitration using a form of distributed parallel contention[6]. The MCA includes a fairness feature which helps ensure that any bus master can obtain the bus in a finite time. Any bus master also has the option to disable fairness to gain additional, but not exclusive(unless it has the highest priority), use of the bus. This mode of operation is designated linear priority.

2.2 Coherence Protocols

In a shared-memory multiprocessor environment, private caches are useful for reducing memory traffic from each processor[21]. A cache coherence protocol allows individual processors to manipulate shared data such that any processor will always use the most recent value of that shared data. One class of coherency protocols is directory based in which information about each block of physical memory is stored in a single directory[14]. The directory of block information can be centralized or it can be distributed in smaller pieces to reduce contention.

For a shared-memory bus system it is common to use snooping protocols to ensure cache coherence. In such protocols, each cache controller monitors memory requests on the system bus. The cache write policy helps to determine which memory requests are placed on the bus. The write-through policy requires that each time the processor writes a data value, the processor uses the bus to update the main memory. This causes the processor to stall while waiting for the bus. We may enhance the basic policy by using write buffers and weaker notions of consistency, making write-through a more viable option. However, the study in [2]

has shown that a write-back policy performed thirty percent better than write-through for a small number of processors. A write-back policy allows a single processor to hold the most recent value of a datum until another processor needs the value or its private cache no longer has space to store the value. At this point either the processor updates main memory or another cache obtains the value and will update the shared memory later. Several write-back policies do not allow main memory to provide a memory block if a processor has altered the block in its private cache. This usually requires additional hardware or bus signals for efficient implementation. For example, the Synapse protocol uses a smart memory controller which maintains a bit per memory block to signify block ownership[3]. On the other hand, the Firefly and Dragon protocols use a *Shared Line* bus signal. The write-once protocol uses a bus signal to inhibit the memory from supplying the data[10].

3 Implementation

To study the multiprocessing performance of the Micro Channel architecture, we have developed a simulator of this bus protocol. Our simulator uses the 32-bit mode of the data bus and allows from 1 to 15 processors to request and use the bus. We assume each word transfer on the channel requires 200ns and each arbitration cycle requires 300ns. For the simulations presented here, we assumed each processor accessed a word every 50ns(when it was not waiting for the bus). Each processor has a private cache. The caches use the channel to communicate with each other and main memory. We developed the Micro Channel and cache simulators separately, and then linked the modules together. This modularity gives us the flexibility to modify the component parts rapidly without affecting the overall functionality of the code.

3.1 Micro Channel Simulation

The micro channel simulator allows each processor to interact with the micro channel independently. When a processor needs the bus(and it is eligible to obtain the bus), its bus interface asserts the bus preempt signal. The Central Arbitration Control Point(CACP) recognizes the preempt and asserts the arbitration signal when any burst transfer, or single transfer, completes. Any processor which needs the bus at that point asserts its priority on the arbitration bus. If a processor is not ready to participate when arbitration begins, the simulator does not allow it to participate during that arbitration period. The local arbiters compare their priority to the value on the arbitration bus. If a processor notices a higher priority signal, then the processor releases its low order signals. Only the highest priority bus master continues to assert its full priority when the arbitration completes and the CACP asserts the grant signal.

When a processor receives the grant, it releases the preempt signal and begins its transfer. The processor asserts the burst signal if it is transferring a block of data or it is using a write-through buffer. Even if only one word is in the write-through buffer, the bus interface asserts the burst signal. The simulator does this because the processor could request another bus transfer before the bus interface completes the single transfer. The simulator

uses the fairness feature to allow all processors to have an equal share of bus grants according to their priorities. If another bus master is asserting the preempt bus signal when the current bus master asserts the burst signal, the bus interface remembers it is not eligible to participate in the next arbitration. It will become eligible again only after all other devices release the preempt bus signal. After the transfer is complete and the bus interface releases the burst signal, another arbitration begins if the preempt bus signal is active.

3.2 Cache Simulation

The simulator accepts several parameters to determine the behavior of the cache. To specify properties of the cache, the parameters include:

- unified or separate data and instruction caches,
- the cache associativity,
- the number of blocks per set, and
- the number of words per block.

The cache coherence protocol is specified by these parameters:

- whether the cache is write-back or write-through,
- whether it allocates blocks on writes or only on reads, and
- the size of the write-through buffer.

We used trace files with about one million address references. These trace files may be too short to measure the actual steady state performance for a system with large private caches. To account for this, we ran simulations for both cold start and hot start caches. The hot start cache eliminated the normally heavy bus traffic caused by cold start(compulsory) misses when a process began.

3.3 Integration

For our initial experiments, we simulated multiple processors running in parallel with no inter-process communication. This lack of inter-process communication reduces bus traffic and eliminates cache block invalidations(which itself reduces bus traffic). Thus, we obtain an experimental upper bound on processor performance. The ratio of inter-process communication to useful processor work will be an important factor in determining an upper limit for the number of active parallel processors the channel can easily accommodate.

The user specifies a trace file for each processor on the bus, which is used to drive the cache simulator. In turn, each instantiation of the cache simulator interacts with the micro channel simulation. To support the integration of these two simulators, we used another procedure for bus snooping and cache block invalidations. While one processor has control of the bus, every other processor continues processing its stream of memory references until it becomes blocked waiting to perform a bus transfer. Each time a new address appears on the address bus, for any processor not using the bus, the snooping procedure is called(although the invalidation feature was disabled for the simulations presented here).

4 Results of Simulation

Our initial experiments have involved three simple cache policies: write through, write through with a write buffer, and write back with write allocate. For each of these policies we varied s_x, the size of cache x where 2Kb $\leq s_x \leq$ 64Kb, and the number of processors n with $1 \leq n \leq 15$. For each memory reference which caused a cache miss or write-through, the bus interface requested the bus to begin a bus transfer. The interactions between the bus, caches, and processors were the basis for our analysis.

We ran simulations using five different trace files with lengths ranging from 291,390 to 1,000,002 memory references. These were some of the traces files publicly available with [14]. For all simulations discussed here, we chose one trace file(SPICE with 1,000,001 addresses) and kept these parameters constant: a unified cache with 4 bytes per word, 16 words per block, and 4 blocks per set(4-way set associativity). To simulate the different cache sizes, we varied the number of sets in the cache. Each simulated processor referenced the entire trace file, thus adding more work to the system with each additional processor. This increased the total work to be done by the system. In this way, we could measure the point at which the bus could not efficiently handle more traffic, and the system could handle no more work.

4.1 Measures of Performance

We compare the measurements between the various system configurations to determine those that perform better. One item for comparison is the completion time for the same process performed with different system configurations. Another item we consider is the percentage of cycles that the bus is inactive. Individually, these may be somewhat misleading if we do not consider other statistics also. For example, we noted that using write-through with no buffer had a higher percentage of inactive bus cycles than the percentage given using write-through with a sixteen word buffer. We can account for this when we consider that the no-buffer cache yields a completion time that is ten percent longer than that attained with the sixteen word buffer.

4.2 Memory Access Cost

The average number of processor clock cycles used per memory reference, $\overline{M}$, gives us an estimate of how much each additional processor can increase the overall memory throughput. For the simulations presented here, the number of memory references for a processor, L_i, was the same for all processors i with $1 \leq i \leq n$. We calculate $\overline{M}$ by dividing the number of cycles for all processors to complete, C_n, by the total number of addresses traced in the complete system, $\overline{M} = C_n / \sum_{i=1}^{n} L_i$. The average cycles per memory reference for an individual processor is approximately $n * \overline{M}$.

$\overline{M}$ varies widely between cache sizes, starting at approximately one for a single processor with a large cache to over six with a small cache. As we increase the number of processors, $\overline{M}$ decreases initially and levels off as the channel saturates. The decrease is not noticeable after two to three processors for small caches, and after five to six for larger caches, except with the write-back policy. Figure 1 shows this for a write-through cache with a sixteen word write buffer(all figures depict data obtained for hot start caches). The level curve

shows that the average cycles per reference for an individual cache grows linearly for the saturated bus. For the write-back policy with large caches, $\overline{M}$ continues to decrease slightly even after 13 to 14 processors, where $\overline{M} \leq 1/n$ is the limit. Systems with a large write-back cache approach this limit as shown in Figure 2. Other simulations showed there is little improvement in $\overline{M}$ for cache sizes larger than 64Kb.

4.3 Time to Complete

For each configuration, we measure the time necessary for a single process to complete, C, while it runs in parallel with zero to fourteen other processes. As we increase the number of processors, once the bus transfers saturate the channel, C increases linearly. The increase is determined by the minimum amount of bus time a process requires. The amount of bus time a process requires also increases as we decrease the cache size.

Another measurement related to the completion time is the additional cycles needed for a process to complete after we add another active processor. This value is merely the slope of the segment connecting one point to the prior point in the graph depicting C. Each time we add another processor with more work, we measure the extra time needed, due to bus contention, for the first process to complete. For caches smaller than 16Kb, the additional time is a significant percentage of the single process execution time. After adding a few active processors, the time needed for each additional process remains nearly constant as shown in Figure 3.

4.4 Cycles per Transaction

We also measured the average number of processor cycles used for each transfer of data on the bus, $\overline{T}$. The simulator counts both the total number of words transferred on the bus, T_w, and the number of arbitrations, A, performed during the system simulation. The average is computed as $\overline{T} = T_w/A$. Caches without a write-through buffer have a constant $\overline{T}$ independent of the number of processors using the bus. For caches with write-through buffers, however, $\overline{T}$ increases as the number of processors increases. $\overline{T}$ approaches a maximum after adding several processors as shown in Figure 4. The maximum primarily depends on both cache size and buffer size. With a small cache, $\overline{T}$ is initially high since the higher miss ratio causes additional block reads. This also causes $\overline{T}$ to approach a maximum sooner than a large cache since the processor only partially fills the buffer before a read miss occurs. $\overline{T}$ is initially much smaller for a large cache. The lower number of read misses allows $\overline{T}$ to be closer to the length of a single write-through. This happens since the buffer normally has only a few values when the bus interface obtains the bus and empties the buffer. As the number of processors increases, $\overline{T}$ grows as longer waits between obtaining the bus forces the processor to fill the buffer. Once again, $\overline{T}$ approaches a maximum when the processor no longer can fill the buffer. This happens not only due to a read miss, but also due to the buffer becoming full or the bus interface releasing the bus before the processor reaches another write in the address trace.

4.5 Transaction Latency

We calculated the average bus wait and transfer time for each bus use for all processors in the system, $\overline{D}$. For a single write, $\overline{D}$ represents the delay time between when a processor changes a datum and when then rest of the system sees the change. We plotted a similar graph, shown in Figure 5, based on each bus request(read block, write block, or write through). Each bus request causes a bus use except while filling a write-through buffer. For those systems without a write-through buffer, these graphs are the same. A large buffer increases the amount of time a processor holds the bus while writing out the buffer. The MCA allows each processor to use the bus for a maximum time before forcing it to release the bus for arbitration. Excessively large buffers require several bus grants to complete all of the writes to memory. As the buffer size increases, $\overline{D}$ increases, but the cycles per bus request decreases. The decrease, however, is not significant for buffers larger than sixteen words.

4.6 Bus Utilization

To obtain an estimate for system bus use, we tracked the number of cycles during which the bus was inactive until the first processor completed, I. We used this to calculate the percentage of unused bus time, I_p, while all of the simulated processors were active. $I_p = I/C * 100$. The simulator considers that the bus is idle if no bus master is using the bus or signaling preempt. Experiments indicate that the bus saturates with five active processors for write-through caches. This is almost independent of buffer size or whether the cache is hot or cold. With four processors using write-through, the bus spends less than five percent of its time inactive. Using a 32Kb write-back cache, however, I_p does not fall below five percent until thirteen processors are active. Examples of these graphs are given in Figures 6 and 7. I_p decreases as buffer size increases and is approximately twenty-five percent smaller for the cold cache versus the hot cache.

Additionally, we measured the efficiency of the first processor which completed its task, as well as the overall system power. We graphed the system power both as predicted by the first processor's efficiency and as measured after all processes completed. Both graphs show at what point adding another processor only causes the efficiency of every process to decrease such that the overall system power is virtually unchanged. Figures 8 and 9 show two examples of the measured system power.

As we add more processors, the system power increases until the point at which the bus can accommodate no more traffic. The predicted and measured system power correlate well except when the write-through policy is used without a buffer. Initially, the predicted power rises quickly and overshoots the saturated power level before reducing to the measured level. The first processor apparently receives a larger share of bus time than the other processors. This is because most of the bus use is for single word writes. Thus, the highest priority processor may participate in almost every other arbitration period when it needs to perform a series of writes.

4.7 Speedup

In these simulations, we add more work each time we add a processor. To estimate speedup, we measure the execution time t for one process on a sequential machine and assume n processes will take time $n * t$ running one after the other on the same sequential machine. At this point we consider what properties of a sequential machine will make it an appropriate one for comparison. One possibility is to assume that since the sequential machine has only one processor using the bus, there is no time lost for bus arbitration. Another consideration is whether we should compare the simulated system using caches with write buffers to a sequential machine with no write buffer. Here the answer is no, since sequential machines also enjoy the benefits of a write buffer even when there is no contention for the bus[19].

We calculated speedup in three ways to allow for several comparisons. The first method is to consider a uniprocessor with the same size cache, but it does not have a write-through buffer or perform arbitration. We total the time it will take the uniprocessor to complete all processes, executing one at a time. We compare the total to the actual completion time of the last process on the multiprocessor. This curve levels off after two to three processors for a small write-through cache, and after approximately one additional processor for each doubling of the cache size as shown in figure 10. This speedup, S_1, also increases as we increase the buffer size for a write-through cache. The curve, with up to fifteen processors, does not become level for write-back caches larger than 16Kb.

We computed the second speedup, S_2, similarly except the uniprocessor now had the same buffer as one processor in the multiprocessor. S_2 rises quickly until there are three or four active processors, then it begins to decline in the presence of write-through buffers. As we increase the buffer size, the decline is more rapid and the curve levels off sooner.

Out third alternative, S_3, assumes the uniprocessor must also participate in arbitration. S_3, shown in Figures 11 and 12, is similar to S_2, except the values are slightly higher due to the uniprocessor's arbitration time. Figure 11 shows S_3 for a write-through cache with a sixteen word buffer. We should mention that each curve on this graph is individually normalized to its one processor case. In Figure 12 we show the same graph for the write-back cache.

5 Future Work

The simulator has several parameters for specifying a system and evaluating its performance. The simulation results presented in this report are for a unified cache having four-way associativity while varying the number of sets. We are currently considering other configurations including separate instruction and data caches and various levels of associativity.

5.1 Micro Channel Features

There are several MCA features which are of potential use in developing a parallel architecture. One such option is to make use of matched memory data transfers. The matched memory protocol allows the processor to communicate with memory at a faster data rate. Additional pins on an extended connector are required to use this option. We will consider

the effects this has on the ability for other processors to snoop the bus. It is possible that this feature will be useful if we can connect all processors to the additional matched memory pins with the logic required to drive these signals. Also, both the memory and the processor should be able to respond in less time than the default cycle time.

Another method available for speeding up data transfer is to utilize the MCA streaming data protocol. This essentially doubles the data transfer rate by removing the need to place an address on the bus for every transfer. This protocol is useful for block transfers of sequential addresses. After the bus interface sends the beginning address during the first transfer, both the processor and the memory will increment the address for each subsequent transfer until the block transfer is complete. Again, the memory and processor must both agree to follow the streaming data protocol. This streaming data protocol will not have an adverse effect on the ability to snoop and invalidate since the address given identifies the entire block. To further double the transfer rate, the bus interface can multiplex data onto the address signals. This allows the transfer of a double word(64 bits).

5.2 Cache Coherence Protocols

Determining the optimal performance of a write-through cache is relatively simple since it principally depends on the percentage of writes in an address trace. Comparison with this optimum level of performance gives us a basis for measuring cost when choosing a cache larger than a certain size. For a write-back cache, however, it is significantly more complex to specify an upper limit on performance. This depends not only on the percentage of writes, but also on the average number of writes to a block before the cache replaces it. It is affected by such things as block size, cache size, associativity, and locality of reference.

When specifying a write-back cache coherence protocol, it is useful to consider the performance of the techniques used by other protocols. Various methods for handling shared data have been implemented in an attempt to reduce or eliminate unnecessary writes on the bus[8]. A coherence protocol can determine sharing dynamically through the use of state bits or bus signals. As an example, we can enhance the write-once protocol to use this sharing information to reduce the number of initial write-throughs when an item is local to one processor only[3]. Another alternative is to determine sharing statically using compiler techniques. The cache can base decisions to perform a write-through on whether or when it shares a datum. Unshared, or private, blocks can remain dirty in a cache until it is necessary to update the block in main memory.

Another choice that we make for a coherence protocol is between write-invalidate and write-broadcast[20]. Write-invalidate causes every other cache to invalidate its copy of the block, while write-broadcast updates the block in every cache which holds an inconsistent copy of that block. Protocols using write-broadcast do not explicitly implement an invalid state for cache blocks. One way to signify a cache block does not contain usable data is to use a reserved tag value that will not match any valid tag search in the cache. Alternatively, the system can use a bit signifying valid data for each block as its initial state which is unreachable unless it flushes the cache.

The performance of either write-invalidate or write-broadcast depends, to a large extent, on the application program. We will determine if one method is significantly easier to

implement or performs better overall in the MCA environment. The relative merits of invalidation and broadcast, and some enhancements specific to each technique, are considered in [9].

A common scheme to improve microprocessor performance is to include an on-chip cache such as with the Intel 80486 and the IBM 386SLC. These particular processors each have an internal eight kilobyte cache. Many of our simulations show that an 8Kb cache is near the borderline of good cache performance. In many cases, a larger external private cache will greatly reduce bus traffic. A processor with an on-chip cache will affect the choice of certain parameters for the external cache, which should be well matched for optimal performance.

5.3 Multiprocessing

We are also considering weaker models of memory consistency which should decrease bus use, allowing more processors per bus. One weaker form of consistency is used by the release consistency model[17, 18]. This requires only that updates to shared variables be completed before leaving a critical section and unlocking the lock. Flushing a write-through buffer before changing a lock value is one means of accomplishing this.

When dealing with communicating processes, it is often necessary to synchronize[1]. We are evaluating the mechanisms for handling synchronization for a shared-memory multiprocessor. These include various software based synchronization mechanisms, as well as hardware alternatives such as special memory cards for maintaining locks. We are studying the relationships between synchronization, bus use, and cache coherence necessary to obtain good processor efficiency. Hardware mechanisms are also useful to improve the support for heavily used software-based locks[11]. These hardware mechanisms should be implemented on an expansion card without modifying the basic system.

The current simulations assume the cache is accessed every processor cycle to locate the next memory reference. We are in the process of instrumenting a PS/2 to measure processor and channel activity while executing various user applications. This will provide a more appropriate delay period between memory accesses to the cache for the simulator. We will obtain combined instruction and data traces of user applications executing on the PS/2. These traces will be uniprocessor memory reference streams which do not translate directly into multiprocessor traces. Similarly, traces obtained on one multiprocessor do not always translate directly into traces for another multiprocessor[15, 16]. The timings and patterns in the PS/2 traces, however, can be used to create separate synthetic traces for processors in the multiprocessor environment we are evaluating.

The data presented in this report is for separate processes which do not share data(bus snooping is not active). One of the areas we will be investigating next is the effect of various amounts of shared references. We will generate a reference stream that includes a specified percentage of shared references, with other parameters to control the reference behavior. We will refine the shared reference model as we study it to generate more realistic behavior(exhibiting specific forms of locality).

6 Conclusions

Many of our simulations have been useful in examining the performance of cache coherence protocols on the Micro Channel architecture. We now consider the simulation results in discussing these three simple protocols in general.

6.1 Write Through Cache

Each write in the memory reference stream causes the processor to stall while waiting to obtain the bus and update main memory when using the write-through coherence protocol. Our simulations have shown that a write-through requires a minimum of ten processor cycles to arbitrate for the channel and perform the transfer. At hot spots, this means a processor could spend 150 cycles(for a system with 15 processors) to perform each write.

One optimization is to unblock a processor as soon as it receives a bus grant. The processor would be able to reference data in the cache while its bus interface performed the transfer. This reduces the blocked time to six cycles for the single processor case. However, for a heavily contested bus, this optimization alone would have only a small effect on the overall system.

6.2 Write Through Cache with a Write Buffer

Adding a write-through buffer decreases process run time in at least two ways. The first is that some of the time spent waiting for the bus and performing writes on the bus overlaps the instruction execution since the processor no longer has to stall for every write. The second reason is that a processor performs more transfers for each arbitration, therefore decreasing the number of arbitrations performed and amortizing the cost of each arbitration over several writes. Since each processor now holds the bus for a longer period, the average time to acquire the bus is longer(although still limited by the enforcement of MCA rules). Fortunately, the processor does not always stall before it acquires the bus. A more useful number to consider may be the total number of stalled cycles versus the total number of requests. For a cache without a write-through buffer, these two averages are the same since the number of arbitrations is equal to the number of requests. When a write-through buffer is present, the number of requests can be significantly larger than the number of arbitrations, since a single arbitration satisfies several requests. When a processor completes its task, its bus interface continues to request the bus until the write-through buffer is empty.

6.3 Write Back Cache

Using a write-through cache coherence protocol, with a large cache, can saturate the bus with fewer than six active processors(depending on the percentage of writes). In contrast, a write-back policy, with fifteen active processors, can leave the bus inactive for 10 to 50 percent of the cycles. This, however, is only while all processors are still active, and none have begun to flush the possibly large number of dirty blocks remaining once the process completes. Table 1 shows the number of dirty blocks left in the cache of a completed process for our simulations. The number of dirty blocks B increased as the cache size increased.

Table 1: A large cache employing the write-back policy requires a noticeable percentage of a programs execution time to flush the cache of dirty blocks.

Cache Size	Dirty Blocks	Extra Cycles	Percentage Time Increase
2 Kb	8	9323	0.01 percent
4 Kb	14	16060	0.04 percent
8 Kb	61	66651	0.34 percent
16 Kb	96	93517	1.50 percent
32 Kb	173	127993	5.72 percent
64 Kb	217	56086	4.79 percent

For each cache size, we have calculated an estimate for the extra time needed to write these blocks back to main memory, $T_{extra} = B * \overline{D} * (I_p/100)$. Using these values, we show the estimated percentage of time the bus interface would extend the completion time of that single process while the cache is being flushed.

For a small cache(2Kb), the write-back policy can perform more poorly than a simple write-through policy. This, of course, depends on the average number of writes that occur to a single block before it is written back. The larger this average is, the more favorable it is to use a write-back cache. The relative performance also depends on the overhead caused by arbitration. A large cost due to arbitration also favors the write-back policy which often requires fewer arbitrations. For these reasons, the write-back cache immediately shows better performance as the cache size increases. In fact, an extremely large simple write-through cache performed no better than a modestly sized write-back cache(approximately 16Kb) during our simulations.

7 Acknowledgments

The authors gratefully acknowledge that this work was supported in part by a grant from IBM Corporation.

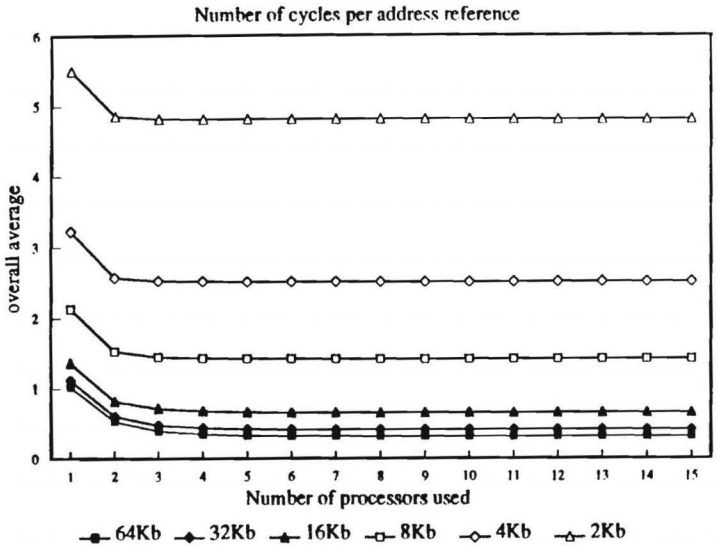

Figure 1: Write-through cache with a sixteen word write buffer.

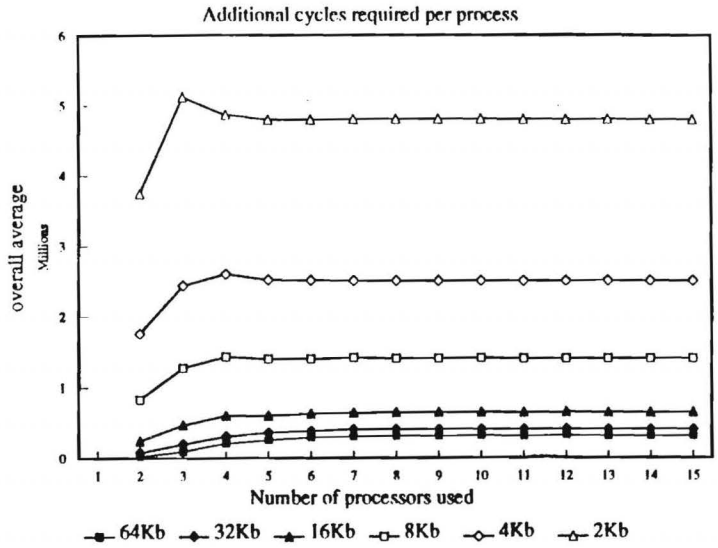

Figure 3: Write-through cache with a sixteen word write buffer.

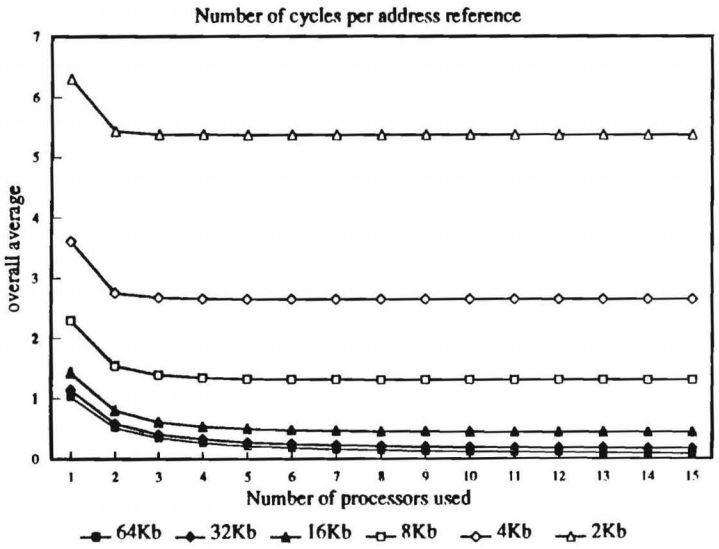

Figure 2: Write-back cache using write-allocate.

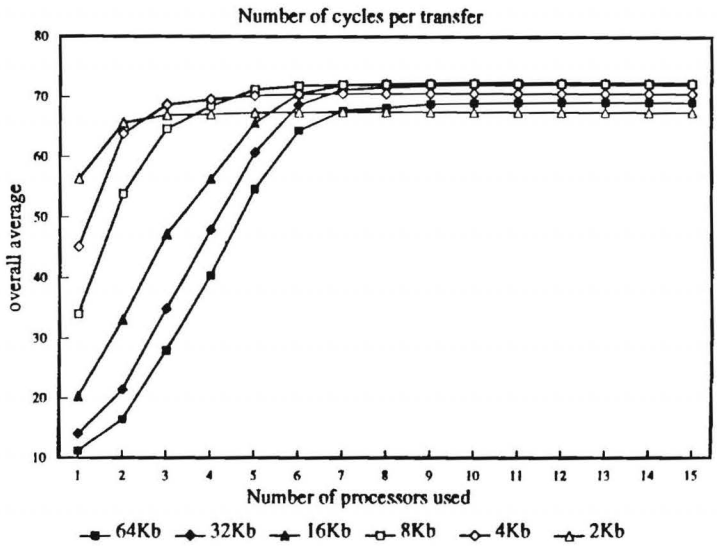

Figure 4: Write-through cache with a sixteen word write buffer.

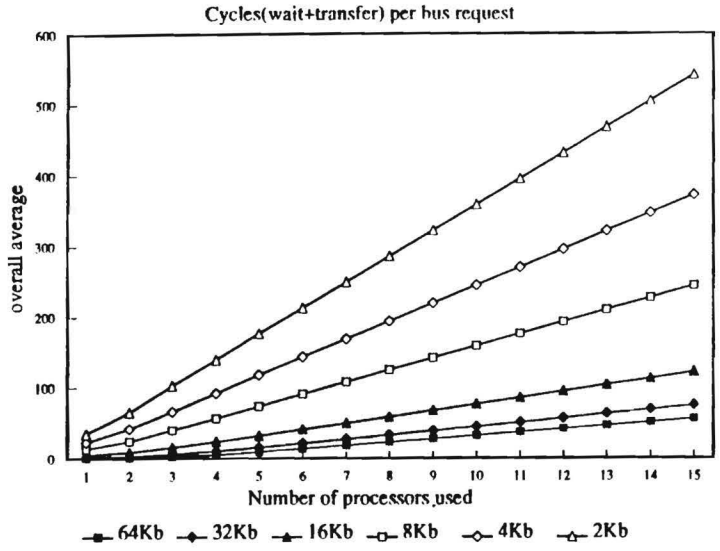

Figure 5: Write-through cache with a sixteen word write buffer.

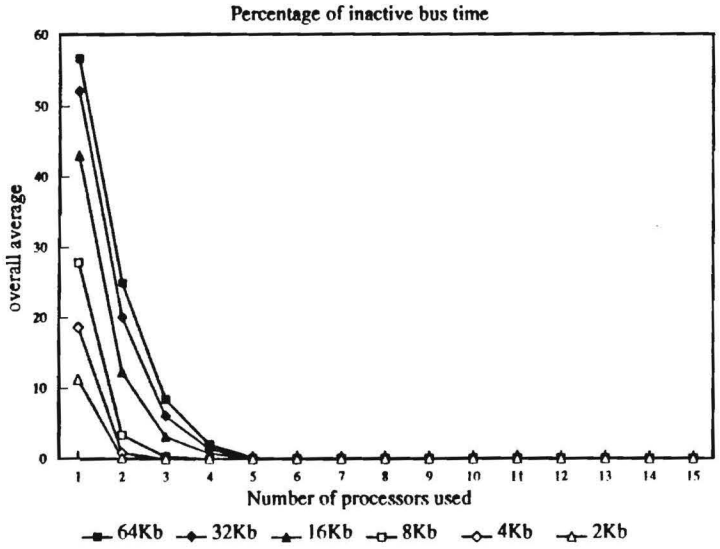

Figure 6: Write-through cache with a sixteen word write buffer.

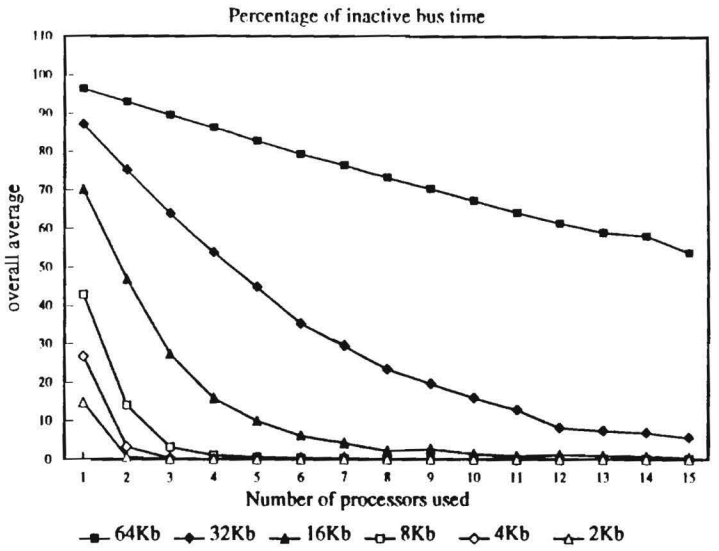

Figure 7: Write-back cache using write-allocate.

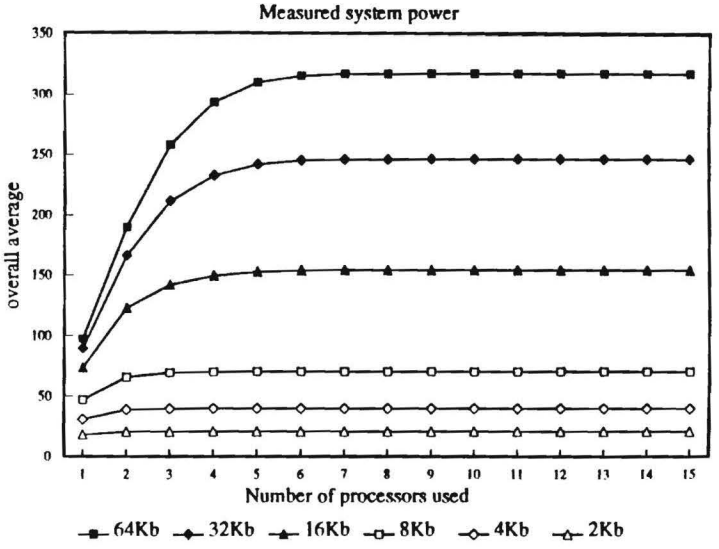

Figure 8: Write-through cache with a sixteen word write buffer.

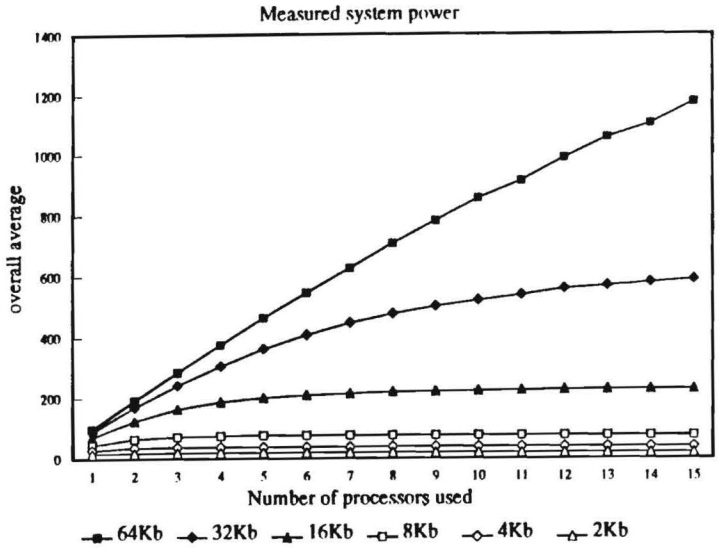

Figure 9: Write-back cache using write-allocate.

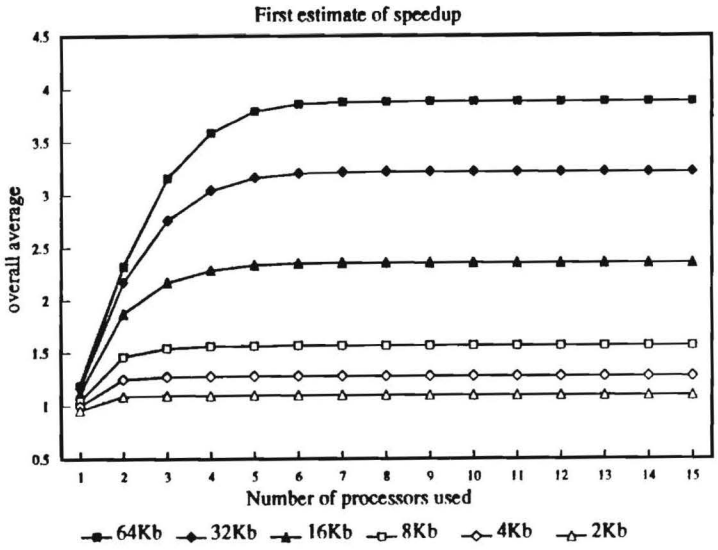

Figure 10: Write-through cache with a sixteen word write buffer.

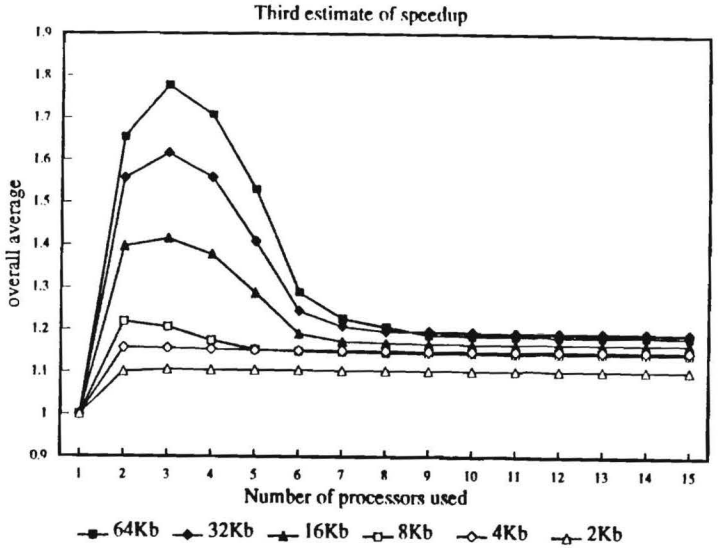

Figure 11: Write-through cache with a sixteen word write buffer.

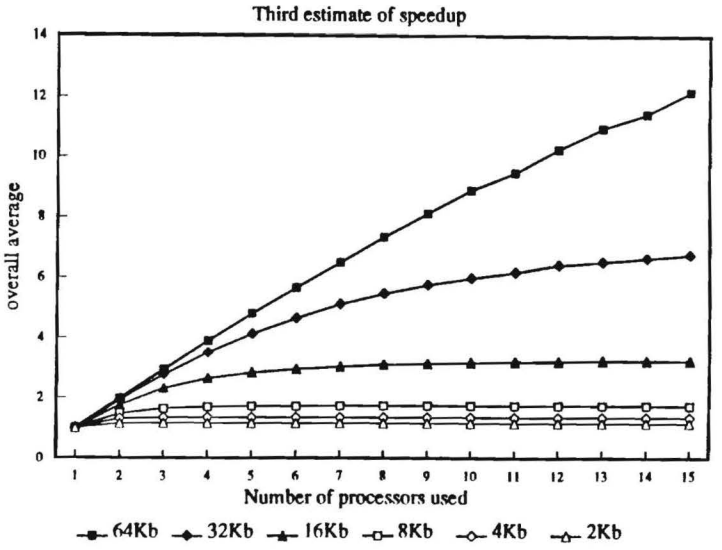

Figure 12: Write-back cache using write-allocate.

References

[1] G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming", *Computing Surveys*, pp. 3–43, Mar 1983

[2] M. Annaratone and R. Ruhl, "Performance Measurements on a Commercial Multiprocessor Running Parallel Code", *16th Annual International Symposium on Computer Architecture*, pp. 307–314, 1989

[3] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, pp. 273–298, Nov 1986

[4] H.B. Bakoglu, G.F. Grohoski, and R.K. Monotoye, "The IBM RISC System/6000 Processor: Hardware Overview", *IBM Journal of Research and Development*, pp. 12–22, Jan 1990

[5] N. Boudette, "586 Processor Card Due for PS/2 Models 90 and 95", *PC Week*, p. 6, 17 Feb 1992

[6] J. DiGiacomo, *Digital Bus Handbook*, Brady, chapters 5 and 15-18, 1990

[7] M. Dubois and J.-C. Wang, "Shared Data Contention in a Cache Coherence Protocol", *Proceedings of the International Conference on Parallel Processing*, pp. 146–155, 1988

[8] S.J. Eggers, and R.H. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation", *15th Annual International Symposium on Computer Architecture, Conference Proceedings*, pp. 373–382, 1988

[9] S.J. Eggers, and R.H. Katz, "Evaluating the Performance of Four Snooping Cache Coherency Protocols", *Conference Proceedings - Ann. Sym. on Comp. Arch.*, pp. 2–15, 1989

[10] J.R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", *Proceedings of the 10th Ann. Int. Sym. on Comp. Arch.*, pp. 124–131, 1983

[11] G. Graunke and S. Thakkar, "Synchronization Algorithms for Shared-Memory Multiprocessors", *Computer*, pp. 60–69, Jun 1990

[12] C. Heath, "The Ins and Outs of IBM's New Bus", *Mini-Micro Systems*, pp. 135–137, Jul 1987

[13] C. Heath and W.L. Rosch, *The Micro Channel Architecture Handbook*, 1990

[14] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, 1990

[15] M.A. Holliday, "Techniques for Cache and Memory Simulation Using Address Reference Traces", *Int. Journal in Computer Simulation*, pp. 129–151, 1991

[16] M.A. Holliday and C.S. Ellis, "Accuracy of Memory Reference Traces of Parallel Computations in Trace-Driven Simulation", *IEEE Trans. on Par. and Dist. Systems*, pp. 97–109, Jan 1992

[17] D. Lenoski *et al.*, "Design of Scalable Shared-Memory Multiprocessors: The DASH Approach", *35th IEEE Comp. Soc. Int. Conf. on Intellectual Leverage*, pp. 62–67, 1990

[18] D. Lenoski *et al.*, "The Stanford Dash Multiprocessor", *Computer*, pp. 63–79, Mar 1992

[19] A.J. Smith, "Cache Memories", *Computing Surveys*, pp. 473–530, Sep 1982

[20] P. Stenstrom, "A Survey of Cache Coherence Schemes for Multiprocessors", *Computer*, pp. 12–24, Jun 1990

[21] C.K. Tang, "Cache System Design in the Tightly Coupled Multiprocessor System", *AFIPS National Computer Conference*, pp. 749–753, 1976